THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Modular Normalization with Types

Nachiappan Valliappan

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Modular Normalization with Types

Nachiappan Valliappan

ISBN 978-91-7905-850-0
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5316
ISSN 0346-718X

Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Printed by Chalmers Reproservice,
Gothenburg, Sweden, 2023

Modular Normalization with Types
Nachiappan Valliappan
Department of Computer Science & Engineering
Chalmers University of Technology

Abstract

With the increasing use of software in today's digital world, software is becoming more and more complex and the cost of developing and maintaining software has skyrocketed. It has become pressing to develop software using effective tools that reduce this cost. Programming language research aims to develop such tools using mathematically rigorous foundations. A recurring and central concept in programming language research is *normalization*: the process of transforming a complex expression in a language to a canonical form while preserving its meaning. Normalization has compelling benefits in theory and practice, but is extremely difficult to achieve. Several program transformations that are used to optimise programs, prove properties of languages and check program equivalence, for instance, are after all instances of normalization, but they are seldom viewed as such.

Viewed through the lens of current methods, normalization lacks the ability to be broken into sub-problems and solved independently, i.e., lacks *modularity*. To make matters worse, such methods rely excessively on the syntax of the language, making the resulting normalization algorithms brittle and sensitive to changes in the syntax. When the syntax of the language evolves due to modification or extension, as it almost always does in practice, the normalization algorithm may need to be revisited entirely. To circumvent these problems, normalization is currently either abandoned entirely or concrete instances of normalization are achieved using ad hoc means specific to a particular language. Continuing this trend in programming language research poses the risk of building on a weak foundation where languages either lack fundamental properties that follow from normalization or several concrete instances end up being repeated in an ad hoc manner that lacks reusability.

This thesis advocates for the use of type-directed *Normalization by Evaluation* (NbE) to develop normalization algorithms. NbE is a technique that provides an opportunity for a modular implementation of normalization algorithms by allowing us to disentangle the syntax of a language from its semantics. Types further this opportunity by allowing us to dissect a language into isolated fragments, such as functions and products, with an individual specification of syntax and semantics. To illustrate type-directed NbE in context, we develop NbE algorithms and show their applicability for typed programming language calculi in three different domains (modal types, static information-flow control and categorical combinators) and for a family of embedded-domain specific languages in Haskell.

Keywords: programming language theory, normalization, type systems

Acknowledgments

My research at Chalmers has been a constant tug of war between what I would like to do and what needs to be done. Should I follow the elegant path of theoretical pursuit? Or must I solve today's problems and push the boundaries of current technology? These questions, in addition to my wide range of interests, have often left me conflicted and occasionally in a state of despair. I cannot imagine advising me being an easy job, and yet Alejandro Russo has been patient and supportive at all the times when it mattered the most. I am grateful for his advice and guidance.

This thesis began with Andreas Abel handing me his notes on Normalization by Evaluation (NbE), the topic of this thesis, from his desk while declaring "these have been waiting for you". I am deeply grateful to Andreas for suggesting NbE and for all the recommendations and technical advice he has offered me over the years.

I am very fortunate to have had Carlos Tomé Cortiñas and Fabian Ruch as my peers, collaborators and friends over the course of developing this thesis. They have helped me refine my intuition-driven approach and I have benefited immensely from the countless hours we have spent together discussing technical and philosophical matters. Fabian has been an unofficial technical advisor since the inception of this thesis, and I am thankful for all his suggestions and enthusiasm towards NbE.

The programming languages community is a wonderful lot of passionate people and I am glad to be a part of it. Dominic Orchard reached out to me at a critical moment when I was doubtful about completing this thesis, and I am grateful for his advice in the discussion that followed. Sam Lindley and Ohad Kammar have taken a considerable interest in both the development of this thesis and my career as a researcher. Sandro Stucki, Thierry Coquand, Graham Leigh and several anonymous reviewers have offered valuable comments and suggestions on several parts of this thesis. I am indebted to all of them for their advice, criticism and encouragement.

Without the unwavering care and support of my parents, I would not be in a position of luxury where I dedicate my time only to things I am interested in. My friends and colleagues at Chalmers—including Matthí, Agustín, Abhiroop, Irene, Jeremy, Robert, Ivan, Mohammad, Elisabet, Alejandro, and Benjamin—have made this journey an enjoyable experience. Without the early encouragement of Murali Krishnan and Richa George, none of this would have happened, and without the enjoyable experiences with my friends, none of this would be worth it. Thank you!

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

Nachi. May 5, 2023.

Contents

Overview

Papers

Overview

I

Introduction

The thesis underlying this bundle of papers, henceforth called a *thesis*, is:

> *Type-directed Normalization by Evaluation is a solution to the problem of developing modular normalization algorithms that are robust to extension.*

Normalization is a broad term used for the process of transforming a program into a canonical shape while preserving its meaning. The objective of normalization can be fundamental (e.g., checking program equivalence or proving a property of a program) or more practical (e.g., optimizing performance or analyzing a program). By "modular" normalization, I mean the ability to decompose a normalization algorithm into independent modules that can be reused under different circumstances.

Normalization algorithms are currently implemented by rewriting the syntax of a given program in accordance with certain *reduction* rules. For example, the following reduction rule specifies that an expression $0 + x$ can be rewritten to x.

$$0 + x \;\mapsto\; x$$

With sufficient reduction rules and sophisticated rewriting strategies, normalization can be achieved even for complex languages. Rewriting techniques are neither the enemy nor an ally of this thesis, but the difficulty with normalizing by rewriting syntax lies in its very nature: it is a process sensitive to the syntax of the language. When the syntax of the language is modified or extended, a rewriting algorithm may need to be revisited entirely. The goal of this thesis is to develop modular normalization algorithms that are robust to modification and extension.

To achieve its goal, this thesis[1] employs a normalization technique known as *Normalization by Evaluation* (NbE) in combination with the *types* of a language. NbE avoids rewriting and instead normalizes a program by evaluating it in a suitable semantic domain. NbE provides an opportunity for a modular implementation of normalization by decoupling the syntax of a language from its semantics. Types further this opportunity by allowing us to dissect a language into isolated fragments, such as functions and products, with an individual specification of syntax and semantics. Explaining the sorcery of NbE and illustrating its potential in the presence of types for implementing modular normalization algorithms for well-typed functional programming languages is the main non-technical contribution of this thesis.

[1] an extended version of my licentiate thesis [38]

I.1 Why Normalization Matters

In the design and implementation of programming languages, normalization is a recurring concept of central importance. The main benefit of normalization lies in its ability to reduce infinitely large equivalence classes of terms identified by their semantics to their normal forms, thus vastly reducing the set of terms that we must take into consideration while reasoning about the language. In programming languages, normalization may have several objectives:

- *Checking program equivalence*: How do we know if the integer expressions $2 + 2 * (x - 1)$ and $4 * (x - 1)$ are equal? We can normalize them to $2 * x$ and $(4 * x) - 4$ respectively, and observe that they are not equal unless $x = 2$. Normalization is widely used to check the equivalence of programs and proofs in the implementation of dependently typed languages and proof assistants.

- *Implementing program optimization*: Normalization can be used to optimise a program. The integer expression $2 + 2 * (x - 1)$ contains the unnecessary overhead of evaluating known arithmetic operations on literal numbers, and can be optimally replaced by $2 * x$ without changing its meaning.

- *Proving properties of complex type systems*: Type systems enable the detection and prevention of errors in a program before executing it by associating every expression in the language with a type. For example, the type assignment `2 : Int` denotes that the expression literal `2` has the integer type `Int`. The integrity of a complex type system lies within its ability to correctly associate a value to its expected type, and not, for example, incorrectly associate a string literal `"hello"` to the type `Int`. This property, called *canonicity*, can be proved with normalization by showing that canonical forms of all (closed) expressions with type `Int` are in fact integers.

- *Proving completeness of semantic specification*: How do we know that a specification of the semantics of a language is complete? That is, how do we know that we have not missed an intended equivalence between two expressions in a language such as $(x + y) * z \approx (x * z) + (y * z)$? Normalization allows us to prove completeness with respect to a semantic model of the language that defines the expected specification, and thus allows us to gain confidence in its completeness with respect to the model.

Normalization is also prevalent in other areas of logic and computer science. For example, in formal logic, normalization is used to prove meta-theoretic properties of a proof system such as logical consistency and the subformula property. In formal verification, normalization is used to convert a logical formula to a normal form for the purpose of deciding its truth. Similarly normalization is also used in databases to eliminate data redundancy and improve the integrity of data in a database. This thesis is developed in the context of programming languages, particularly well-typed functional programming languages, but it may have applications beyond this area.

I.2 The Sorcery of Normalization by Evaluation

This subsection gives an introduction to the essence of NbE by illustrating the implementation of an NbE algorithm for an extremely simple language: arithmetic expressions consisting of the addition of natural numbers. For this purpose, we encode the natural numbers 0 as *Zero*, 1 as *Succ Zero*, 2 as *Succ* (*Succ Zero*), and so on, using a constant symbol *Zero* and a successor function *Succ*. Addition in a given expression can be reduced using one of the two following reduction steps.

$$\begin{aligned} \mathit{Zero} + x &\mapsto x \\ (\mathit{Succ}\ x) + y &\mapsto x + (\mathit{Succ}\ y) \end{aligned}$$

The reduction relation $\mapsto$ specifies how an expression must be reduced to another expression. Using this specification, the expression 1 + 2 can be normalized to 3 by reducing it as follows.

$$\begin{aligned} &\mathit{Succ\ Zero} + \mathit{Succ}\ (\mathit{Succ\ Zero}) && (1+2) \\ &\quad\mapsto \mathit{Zero} + \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ\ Zero})) && (0+3) \\ &\quad\mapsto \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ\ Zero})) && (3) \end{aligned}$$

Observe that we rewrite the expression twice before reaching the normal form, which cannot be reduced anymore since none of the reduction steps apply. Complex expressions may need to be rewritten several times before a normal form is reached. Rewriting is the basis for traditional normalization procedures, while NbE, on the other hand, does not involve any rewriting.

NbE achieves normalization in two steps: 1) *evaluating* the expressions in a "host" language, and 2) *quoting* (sometimes called *reifying*) the resulting values back to expressions. Let us implement NbE for our example language using the programming language Haskell as the host.

- *Evaluation:* We implement the first step using an interpreter function called `eval`. This function interprets natural numbers as integers and the addition of natural numbers by addition of integers.

```
eval :: Expr Nat -> Int
eval Zero     = 0
eval (Succ x) = eval x + 1
eval (x + y)  = eval x + eval y
```

- *Quotation:* The second step is to invert the integer values back to natural number expressions, and is implemented by a function called `quote`. This function need not be defined on all integer values, but only on the values that may be returned by `eval`.

```
quote :: Int -> Expr Nat
quote 0 = Zero
quote n = Succ (quote (n - 1))
```

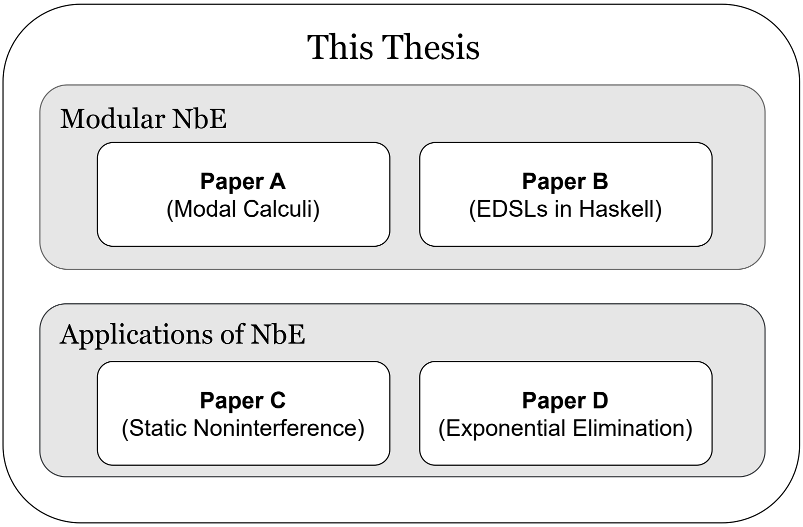

Figure I.1: Outline of this thesis

We implement the normalization procedure by a function `norm` that applies `quote` on the result of `eval`.

```
norm :: Expr Nat -> Expr Nat
norm e = quote (eval e)
```

Observe that an invocation of `norm` on the expression *Succ Zero* + *Succ* (*Succ Zero*) does indeed return its normal form *Succ* (*Succ* (*Succ Zero*)). `norm` uses the ability of Haskell to evaluate the addition of integers to normalize the addition of natural numbers. This function can be extended easily to other arithmetic operators, and, with some care, even to support variables and other unknowns in expressions.

This seemingly simple idea to leverage a host language's evaluation mechanism to normalize expressions extends much beyond arithmetic expressions, and has found a wide range of applications. NbE has been used to achieve normalization results in various programming calculi [2, 7, 9, 18, 24, 30], decide equality in algebraic structures [4], typecheck dependently-typed programming languages [3, 25], and to prove completeness [5, 17] and coherence [10] theorems. NbE algorithms have been observed to yield much faster normalization than their rewriting counterparts [8, 29], and there is also evidence that indicates that it can be used to speed up compilation in optimizing compilers [29].

I.3 Fistful of Problems and This Thesis

This section gives an overview of the problems addressed in this thesis by the papers in the forthcoming chapters. These problems occur independently in different domains, and thus the following subsections may be read in any order. These subsections discuss the interest in these problems (and their domains) and provide an introduction to the corresponding chapters—see Figure I.1 for an outline.

I.3.1 Fitch-Style Modal Calculi

Modal types In type systems, a *modality* can be broadly construed as a unary type constructor with certain properties. Type systems with modalities have found a wide range of applications in programming languages to capture and specify properties of a program in its type. For example, in language-based security, a field dedicated to developing secure programming languages, a substantial number of languages use modalities to ensure sensitive data is not leaked to an unauthorized principal [27]. Using a modal type `Secret Int`, the programmer can indicate via the modality `Secret` to the type system that the underlying integer value must be kept a secret. The type system automatically tracks the flow of this integer in the program and prevents the need for a careful and error-prone manual analysis. Modal type systems offer a form of lightweight and low cost alternative to formal verification of programs for preventing software errors since type systems are a familiar abstraction used widely in mainstream programming languages.

The design and implementation of modal type systems for various applications is a vibrant area of ongoing research. Different applications may demand different modal operations, which means there can be several different kinds of modalities. The *necessity* modality is one such modality that has found applications in modelling purity in an impure functional language [13], confidentiality in information-flow control [32], and binding-time separation in partial evaluation and staged computation [20].

Fitch-style modal calculi Fitch-style modal lambda calculi [12, 16, 31] feature necessity modalities in a typed lambda calculus by extending the typing context with a delimiting "lock" operator. The characteristic lock operator simplifies formulating calculi that incorporate different modal operations and these calculi have excellent computational properties. Each variant demands, however, different, tedious and seemingly ad hoc treatment to prove meta-theoretic properties such as normalization. In Chapter A, we identify the *possible-world* semantics of Fitch-style calculi and use it to develop normalization. The possible-world semantics enables a modular implementation of normalization for various Fitch-style calculi by isolating their differences to a specific parameter that identifies the modal fragment. We showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi based on different interpretations of the necessity modality in programming languages, such as capability safety, noninterference and a form of binding-time correctness.

I.3.2 Embedded Domain-Specific Languages

Overview An *embedded* domain-specific language (eDSL) is an implementation of a domain-specific language (DSL) as a library in a host language. Implementing a DSL as an eDSL offers two main advantages:

- The programmer can leverage the features of the host, typically a more powerful general purpose programming language, to write programs in the eDSL.
- Developing an eDSL compiler requires much lesser effort than building a dedicated DSL compiler, since the host language's compiler can be reused for standard compilation phases such as lexical analysis, parsing and type-checking.

The tradeoff, however, is that programming an eDSL may require some familiarity of the host language.

Let us consider the example with arithmetic expressions again. The following library functions in Haskell constitute an eDSL to write simple arithmetic expressions.

```
val :: Int -> Expr Int
(+) :: Expr Int -> Expr Int -> Expr Int
(*) :: Expr Int -> Expr Int -> Expr Int
```

Using these functions, we can write the expression `1+2` as `val 1 + val 2`.

Suppose that we would like to write an expression x^n that represents the n-th power of an expression x, for some known non-negative integer n. How should we do this when exponentiation is not a primitive function provided by the eDSL? If this were a mere DSL, we would write $x * x * x$ for x^3, for example, since multiplication is provided. In an eDSL, however, we can take this a step further to write a generic `power` function that generates this expression automatically for an arbitrary integer n.

```
power :: Int -> Expr Int -> Expr Int
power n x = if (n <= 0) then x else (x * (power (n - 1))
```

Using the `power` function, we may write `power 8 x` for x^8 instead of $x * x * x * x * x * x * x * x$. The former variant is concise, less error-prone and also makes it easy to modify and reuse code.

Notice that the definition of the power function uses Haskell's features such as conditionals (`if ...`), comparison (`n <= 0`) and function recursion (`power (n - 1)`). Even if the eDSL does not implement these features natively, we are able to use them to write expressions. EDSLs make it easy to derive additional functionality by leveraging those the host language. EDSLs, specifically in Haskell, have found a wide range of applications: hardware description [11], digital signal-processing [6], runtime verification [21, 35], parallel and distributed programming [14, 23], GPU programming [15]—and the list goes on.

Compiling EDSLs In an eDSL program we may think of a value of type `Int` as a *static* integer that is known at compile-time, and a value of type `Expr Int` as a *dynamic* integer that is known only at runtime. This *stage separation* of values as static and dynamic corresponds to a manual form of *binding-time analysis* in partial evaluation [26], and presents an opportunity to exploit Haskell's execution mechanism to evaluate static computations in an eDSL program.

Though separation of stages enables the programmer to manually specify those parts of an eDSL program that must be evaluated by Haskell, it also burdens them to maintain multiple variants of the same program. In addition to `power` function defined above, we may also desire the several variants of the exponentiation function as follows, each corresponding to a different separation of stages for its arguments and result.

```
power0 :: Int -> Int -> Int
power1 :: Int -> Expr Int -> Expr Int
power2 :: Expr Int -> Int -> Expr Int
power3 :: Int -> Int -> Expr Int
...
```

NbE offers a modular solution to this problem by making specialization automatic, without the need for manual stage separation. Chapter B shows that typed NbE is particularly well-suited for specializing eDSL programs in Haskell given the natural reliance on a host language. We argue that existing techniques for embedding DSLs in Haskell (e.g., [37]), which may at first seem somewhat ad hoc, can be viewed as instances of NbE after all.

I.3.3 Language-Based Security

Information-Flow Control. Information-Flow Control (IFC) is a language-based security enforcement technique that guarantees the confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called noninterference. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by the program from its output.

Proof by Normalization. To prove that an IFC system ensures noninterference, we must show that the public output of secured programs remain unaffected by variations in its secret inputs. If the output remains unaffected by a given input, then it must be the case that it does not depend on the input to compute the output—thus ensuring that the attacker could not possibly learn about the secret inputs. Such programs may *refer* to the secret input in its body, but they must not *use* it to compute the public output.

Chapter C proposes a new syntax-directed proof strategy to prove noninterference for well-typed programming calculi that enforce static IFC. The key idea of this chapter is to use normalization to eliminate any unnecessary input references in a program, leaving behind references that are only absolutely necessary to compute the result. Noninterference is then proved by ensuring that no public output depends on a reference to a secret input in the normal form of a program—a task that is much simpler than most semantics-based proof techniques. This technique is illustrated for a model of the terminating fragment of the seclib library [36] in Haskell, which is a simply-typed lambda calculus extended with IFC primitives.

I.3.4 Categorical Combinators

Combinator Calculi. Combinators can be understood as program building blocks which can be assembled in various ways to construct programs. In functional programming, a combinator is a primitive higher order function, which can be applied to and composed with other combinators to build more complex functions. Unlike programming languages based on the lambda calculus, combinators lack a notion of variables. In practice, this means that programming using combinators can be an unbearable task and should probably be avoided at all costs. But then, why care about combinators at all?

> *"...roughly λ-calculus is well-suited for programming, and combinators (of Curry, or those introduced here) allow for implementations getting rid of some difficulties in the scope of variables."*
> —P.-L. Curien (1985, Typed Categorical Combinatory Logic)

The output of a function in the lambda calculus is computed using a process known as β-reduction. The primary difficulty with β-reduction lies in its very definition: the output of a function $\lambda x.b$ for some input i is computed by *substituting* all occurrences of the argument variable x, in the body of the function b, with the actual input i. This statement is succinctly captured by the β-rule:

$$(\lambda x.b)i \;\mapsto\; b[i/x]$$

This rule states that a function $\lambda x.b$ when applied to an argument i, can be reduced to a simpler term $b[i/x]$, which is the result of substituting all occurrences of x with i in the body of the function b. Although substitution readily appeals to the intuition of replacement, there are a number of auxiliary conditions that must be checked before the actual replacement of x with i. For this reason, substitution has long had a reputation for being notoriously difficult to implement and reason about.

Combinators, on the other hand, avoid the need for substitution by disallowing variables entirely. Instead, they adopt a style of reduction that relies on simply "shifting symbols". The (categorical) combinator equivalent of the β-rule is, what I like to call, the *exponential elimination* rule:

$$\mathit{apply} \circ \langle \Lambda b, i\rangle \;\mapsto\; b \circ \langle \mathit{id}, i\rangle$$

This rule reads as: the application (*apply*) of a function (Λb) to an argument (i) can be reduced to a composition of the body (b) with its input (i) in an appropriate manner. The operator $\_ \circ \_$ denotes the sequencing, or composition, of two combinators and $\langle \_, \_ \rangle$ denotes the coupling, or pairing, of two combinators. We shall return to the specifics of this rule in a later chapter, but simply observe here that it does not use the substitution operation on the right-hand side, and that the body of the function (b) remains unmodified.

The absence of substitution, an external operation, means that we need not impose additional correctness criteria over the computation rules—which is great news for formal reasoning! In essence, the very characteristic of combinators that makes them impractical for programming also makes them amenable to implementation and reasoning: the lack of variables.

Categorical Combinators. Categorical combinators are combinators designed after arrows, or *morphisms*, in category theory. They were introduced by Pierre-Louis Curien as an alternative to the SKI combinator calculus to implement functional programming languages.

The primary motivation behind categorical combinators appears to be two-fold: 1) to faithfully simulate reduction in lambda calculus without the difficulty of variable bindings, and 2) to establish a syntactic equivalence theorem between the lambda calculus and the categorical model underlying the combinators—namely, the *(free) cartesian closed categories*. Categorical combinators offered an appealing alternative to Church's more popular SKI combinator calculi, since their design is based on a semantic model. This means that the reduction rules of the combinators arise naturally from the model rather than having to be imposed.

> *"...categorical combinatory logic is entirely faithful to βreduction where [Curry's SKI] combinatory logic needs additional rather complex and unnatural axioms to be..."*
> —P.-L. Curien (1986, Categorical Combinators)

Categorical combinators were used to formulate the *Catergorical Abstract Machine* (CAM) [19], which was used to used to implement early versions of Caml—the predecessor of the OCaml programming language. Later versions of Caml, however, did not use CAM due to performance issues and difficulty with optimizations [2]. Despite its failure in use for compiling a programming language in practice, the ease of formulating an abstract machine for categorical combinators (noted in [1]) seems to have influenced several variants of CAM, an example of which is the Linear Abstract Machine [28].

In recent times, variants of (what appear to be) categorical combinators have reappeared in practical applications. They have been used to compile Haskell code using user-defined interpretations [22] and in the development of a language for executing smart contracts on the blockchain [34].

Exponential elimination. Exponentials are the equivalent of higher-order functions in categorical combinator calculi. The runtime representation of an exponential is a *closure*, a value accompanied by an environment. Adding support for closures complicates the implementation of the abstract machine, and makes certain static analyses difficult [39]. In [22], exponentials narrow the domain of target interpretations that are supported by the compiler.

The exponential elimination rule from earlier indicates that exponentials can be eliminated in a specific case. This makes us wonder: can exponentials be eliminated statically by applying this rule repetitively on a program? This would solve both the above problems. Without a careful analysis, however, it is difficult to answer this question, since there may be interactions with other rules in the calculus that prevent exponential elimination rule from being applied.

Chapter D shows that exponential elimination can be achieved for categorical combinators with sums and products, in the presence of a special *distributivity* combinator that distributes products over sums. The ability to erase the equivalent of

[2] https://caml.inria.fr/about/history.en.html

higher-order functions in functional calculus (known as *defunctionalization*) is not news [33], but the distributivity requirement is a somewhat surprising insight. A technical challenge faced by this result is the presence of the empty and sum types, both of which are known for making normalization notoriously difficult.

II

Statement of contributions

This thesis is a bundle of articles published at different venues focused on programming language research. The initial conception, overall development and writing of all these articles were led by me. This chapter outlines my individual contributions to their technical development alongside a listing of their abstracts.

A Normalization for Fitch-Style Modal Calculi

Nachiappan Valliappan, Fabian Ruch, Carlos Tomé Cortiñas

Fitch-style modal lambda calculi enable programming with necessity modalities in a typed lambda calculus by extending the typing context with a delimiting operator that is denoted by a lock. The addition of locks simplifies the formulation of typing rules for calculi that incorporate different modal axioms, but each variant demands different, tedious and seemingly ad hoc syntactic lemmas to prove normalization. In this work, we take a semantic approach to normalization, called normalization by evaluation (NbE), by leveraging the possible-world semantics of Fitch-style calculi to yield a more modular approach to normalization. We show that NbE models can be constructed for calculi that incorporate the K, T and 4 axioms of modal logic, as suitable instantiations of the possible-world semantics. In addition to existing results that handle β-equivalence, our normalization result also considers η-equivalence for these calculi. Our key results have been mechanized in the proof assistant Agda. Finally, we showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi as well as programming-language applications based on different interpretations of the necessity modality.

Statement of contributions I independently mechanized the first Agda prototype using the categorical semantics of Fitch-style calculi and identified the common pattern in the construction of their NbE models. Fabian showed me a connection to possible-world semantics in modal logic that gave a systematic and elegant account of this pattern. This convinced me to factor the construction of the NbE models through possible-world semantics, roughly midway during this development, from which point onwards Fabian and I co-developed the remaining technical results. Carlos helped us explore and understand the applications of these calculi.

Appeared in: *Proceedings of the ACM on Programming Languages Vol 6. ICFP (2022)*

B Practical Normalization by Evaluation for EDSLs

Nachiappan Valliappan, Alejandro Russo, Sam Lindley

Embedded domain-specific languages (eDSLs) are typically implemented in a rich host language, such as Haskell, using a combination of deep and shallow embedding techniques. While such a combination enables programmers to exploit the execution mechanism of Haskell to build and specialize eDSL programs, it blurs the distinction between the host language and the eDSL. As a consequence, extension with features such as sums and effects requires a significant amount of ingenuity from the eDSL designer. In this paper, we demonstrate that Normalization by Evaluation (NbE) provides a principled framework for building, extending, and customizing eDSLs. We present a comprehensive treatment of NbE for deeply embedded eDSLs in Haskell that involves a rich set of features such as sums, arrays, exceptions and state, while addressing practical concerns about normalization such as code expansion and the addition of domain-specific features.

Statement of contributions I developed all the technical results in this paper under the supervision of Alejandro. Sam helped us understand and survey earlier work (some unpublished) that set out to leverage NbE to embed DSLs.

Appeared in: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (2021)*

C Simple Noninterference by Normalization

Carlos Tomé Cortiñas, Nachiappan Valliappan

Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's `seclib` library, using normalization. Our proof follows by straightforward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

Statement of contributions Carlos and I shared the technical development in this work. I constructed most of the NbE model and proved it correct, while Carlos helped me understand, formulate and prove noninterference.

Appeared in: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (2019)*

D Exponential Elimination for Bicartesian Closed Categorical Combinators

Nachiappan Valliappan, Alejandro Russo

Categorical combinators offer a simpler alternative to typed lambda calculi for static analysis and implementation. Since categorical combinators are accompanied by a rich set of conversion rules which arise from categorical laws, they also offer a plethora of opportunities for program optimization. It is unclear, however, how such rules can be applied in a systematic manner to eliminate intermediate values such as *exponentials*, the categorical equivalent of higher-order functions, from a program built using combinators. Exponential elimination simplifies static analysis and enables a simple closure-free implementation of categorical combinators—reasons for which it has been sought after.

In this paper, we prove exponential elimination for *bicartesian closed* categorical (BCC) combinators using normalization. We achieve this by showing that BCC terms can be normalized to normal forms which obey a weak subformula property. We implement normalization using Normalization by Evaluation, and also show that the generated normal forms are correct using logical relations.

Statement of contributions I developed all the technical results in this paper under the supervision of Alejandro.

Appeared in: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (2019)*

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of functional programming*, 1(4):375–416, 1991.

[2] A. Abel and C. Sattler. Normalization by evaluation for call-by-push-value and polarized lambda calculus. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, pages 1–12, 2019.

[3] A. Abel and H. Talk. Normalization by evaluation: Dependent types and impredicativity. *Unpublished. http://www.tcs.ifi.lmu.de/˜abel/habil.pdf*, 2013.

[4] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310. IEEE, 2001.

[5] T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda\!\rightarrow 2$. In *International Symposium on Functional and Logic Programming*, pages 260–275. Springer, 2004.

[6] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE, 2010.

[7] V. Balat, R. Di Cosmo, and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, 2004.

[8] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In *Prospects for Hardware Foundations*, pages 117–137. Springer, 1998.

[9] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. 1991.

[10] I. Beylin and P. Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *International Workshop on Types for Proofs and Programs*, pages 47–61. Springer, 1995.

[11] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. *ACM SIGPLAN Notices*, 34(1):174–184, 1998.

[12] V. Borghuis. *Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus.* PhD thesis, Mathematics and Computer Science, 1994.

[13] V. Choudhury and N. Krishnaswami. Recovering purity with comonads and capabilities. *Proc. ACM Program. Lang.*, 4(ICFP):111:1–111:28, 2020.

[14] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[15] K. Claessen, M. Sheeran, and J. Svensson. Obsidian: Gpu programming in haskell. *Designing Correct Circuits*, page 101, 2008.

[16] R. Clouston. Fitch-style modal lambda calculi. In C. Baier and U. D. Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2018.

[17] C. Coquand. From semantics to rules: A machine assisted analysis. In *International Workshop on Computer Science Logic*, pages 91–105. Springer, 1993.

[18] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[19] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of computer programming*, 8(2):173–202, 1987.

[20] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[21] F. Dedden. Compiling an haskell edsl to c: A new c back-end for the copilot runtime verification framework. Master's thesis, 2018.

[22] C. Elliott. Compiling to categories. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–27, 2017.

[23] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, 2011.

[24] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 151–165. Springer, 2001.

[25] D. Gratzer, J. Sterling, and L. Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[26] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[27] G. A. Kavvos. Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.*, 3(POPL):20:1–20:29, 2019.

[28] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59:157–180, 1988.

[29] S. Lindley. Normalisation by evaluation in the compilation of typed functional programming languages. 2005.

[30] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[31] S. Martini and A. Masini. A computational interpretation of modal proofs. In *Proof theory of modal logic (Hamburg, 1993)*, volume 2 of *Appl. Log. Ser.*, pages 213–241. Kluwer Acad. Publ., Dordrecht, 1996.

[32] K. Miyamoto and A. Igarashi. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security*, pages 187–203, 2004.

[33] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 25–36, 2016.

[34] R. O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120, 2017.

[35] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.

[36] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. *ACM Sigplan Notices*, 44(2):13–24, 2008.

[37] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.*, 44:143–165, 2015.

[38] N. Valliappan. *Be My Guest: Normalizing and Compiling Programs Using a Host Language.* PhD thesis, Chalmers Tekniska Hogskola (Sweden), 2020.

[39] N. Valliappan, S. Mirliaz, E. L. Vesga, and A. Russo. Towards adding variety to simplicity. In *International Symposium on Leveraging Applications of Formal Methods*, pages 414–431. Springer, 2018.